HW/SW Co-designed Processors: Challenges, Design Choices and a Simulation Infrastructure for Evaluation

Rakesh Kumar*, José Cano*, Aleksandar Brankovic†, Demos Pavlou‡,
Kyriakos Stavrou‡, Enric Gibert§, Alejandro Martínez¶ and Antonio González‖
*University of Edinburgh, UK †Intel ‡11pets
§Pharmacelera ¶ARM ‖Universitat Politècnica de Catalunya, Spain

***Abstract*—Improving single thread performance is a key challenge in modern microprocessors especially because the traditional approach of increasing clock frequency and deep pipelining cannot be pushed further due to power constraints. Therefore, researchers have been looking at unconventional architectures to boost single thread performance without running into the power wall. HW/SW co-designed processors like Nvidia Denver, are emerging as a promising alternative.**

However, HW/SW co-designed processors need to address some key challenges such as startup delay, providing high performance with simple hardware, translation/optimization overhead, etc. before they can become mainstream. A fundamental requirement for evaluating different design choices and trade-offs to meet these challenges is to have a simulation infrastructure. Unfortunately, there is no such infrastructure available today. Building the aforementioned infrastructure itself poses significant challenges as it encompasses the complexities of not only an architectural framework but also of a compilation one.

This paper identifies the key challenges that HW/SW co-designed processors face and the basic requirements for a simulation infrastructure targeting these architectures. Furthermore, the paper presents DARCO, a simulation infrastructure to enable research in this domain.

I. Introduction

Microprocessor design has traditionally been and will continue to be driven by high performance requirements. However, due to the slow down in single core (and thread) performance improvement, chip manufacturers now put multiple simple cores onto a single chip instead of making the core more powerful [1]. Such multicore processors improve the overall throughput by executing multiple threads in parallel on different cores. However, the single thread performance suffers because the individual core itself is less powerful. Single thread performance remains of utmost importance for non-parallelizable applications. Furthermore, even for parallel applications the sequential execution of non-parallelizable sections significantly affects the overall performance, as explained by Amdahl's law. Therefore, researchers have been looking at alternative architectures to improve the single thread performance.

This work was done when Rakesh Kumar, José Cano, and Aleksandar Brankovic were with UPC Barcelona; Demos Pavlou, Kyriakos Stavrou, Enric Gibert, and Alejandro Martínez were with Intel Labs; and Antonio González was with both UPC Barcelona and Intel Labs.

Hardware/Software (HW/SW) co-designed processors[1] [3]–[6] are a promising alternative that not only improves the single thread performance but also reduces the power consumption, thereby providing a better performance/watt. These processors employ a software layer that resides between the hardware and the operating system. This software layer dynamically translates the guest ISA instructions to the host ISA, thereby enabling proprietary host ISAs without modifying the software stack. The host ISA is the ISA which is implemented in the hardware, whereas, guest ISA is the one for which applications are compiled. The key idea is to have a simple host ISA to reduce both power consumption and complexity. Furthermore, dynamic binary optimizations are applied to boost the performance.

Nvidia Denver [3] and Transmeta Crusoe [4] are the most representative examples of commercial products based on HW/SW co-designed architecture. Nvidia Denver executes ARMv8 binaries on 7-wide in-order cores to keep the power budget to minimum, and relies on dynamic binary optimizations for performance. Nvidia's benchmark testing showed that the 64-bit CPU outperformed all other announced ARMv8 cores at that time and even matched the performance of low-end versions of Intel's Haswell CPU [3]. Similarly, Transmeta Crusoe executes x86 binaries on a VLIW hardware and also employs dynamic optimization.

Despite its potential, we have yet to see a successful commercial product based on the HW/SW co-design paradigm, as Transmeta Crusoe failed and Nvidia Denver is still under scrutiny. The lack of successful products advocates the need for further research in this domain to realize its full potential. A fundamental requirement to explore and investigate the design choices and trade-offs that these processors offer, is to have a simulation infrastructure. However, there is no simulation infrastructure available currently that provides the capabilities to investigate the challenges in this domain.

Building a simulation infrastructure for HW/SW co-designed processors poses significant challenges. The principal challenge is the complexity of modelling the software layer. The software

[1]The term "HW/SW co-designed processor" is used following the taxonomy provided by [2]. Specifically, they call it "HW/SW co-designed virtual machine" however, we believe the word "processor" is better suited in our context.

layer needs to provide virtually all the functionality of a compilation framework such as translating the guest binary to intermediate representation (IR), optimizing the intermediate code, instruction scheduling, register allocation, generating host code from the IR, etc. in addition to profiling the execution for gathering runtime information and housekeeping. Therefore, a simulation infrastructure for HW/SW co-designed processors can be thought of as encompassing the complexities of a compilation framework in addition to the conventional architectural simulation.

Motivated by the potential of HW/SW co-designed processors and the need for further research in this domain, we have developed a simulation infrastructure, called DARCO, as a first step towards enabling research in this area. DARCO models a processor that profiles, translates, optimizes and executes a guest x86 binary on a RISC host architecture. The key components of DARCO are the guest and host ISA functional emulators, the software layer called Translation Optimization Layer (TOL), timing and power simulators and the accompanying debugging and monitoring tools. TOL is equipped with an interpreter, translator, profiler and a dynamic optimizer that applies a plethora of optimizations to the translated regions. Except for the functional models and power simulator, for which we used modified versions of QEMU [7] and McPAT [8] respectively, all other components are in-house developments. DARCO is a research enabler in HW/SW co-designed processor domain.

The key contributions of the paper include:

- Identifies the principal challenges that HW/SW co-designed processors face and shows how DARCO can be used to address them.
- Identifies the challenges in building a simulation infrastructure for HW/SW co-designed processors.
- Presents DARCO, a complete infrastructure for investigating HW/SW co-designed processors.
- Characterizes the software layer of DARCO and shows that 90% of the guest dynamic code executes with highest optimization level while incurring minimal overhead.

II. HW/SW Co-designed Processors

A HW/SW co-designed processor is a hybrid architecture that leverages HW/SW co-design to couple a software layer to the microarchitectural design of a processor. The basic idea behind these processors is to have a simple host ISA, to reduce power consumption and complexity, with the software layer translating and optimizing the guest binaries for the host ISA. This kind of processors [5], [6], [9] have enticed researchers for more than a decade. Moreover, currently there is a renewed interest from both industry and academia [10]–[13] in the wake of imminent end of Moore's law. In the transistor limited era, the unique capabilities of the software layer will play a key role in performance and energy optimizations.

In general, HW/SW co-designed processors implement a proprietary ISA in hardware to achieve design simplicity and energy efficiency. Therefore, they need to apply binary translation to map the guest ISA on to the host ISA. The binary translation, in general, can be implemented in either hardware or software. For example, modern processors implementing CISC ISA, like x86, implement binary translation in hardware to translate CISC instructions to microcodes dynamically and simplify the execution pipeline implementation [2]. On the other hand, dynamic binary translation in HW/SW co-designed processors is done by the software layer, as shown in Fig. 1. We call this software layer as Translation Optimization Layer (TOL) in this paper.

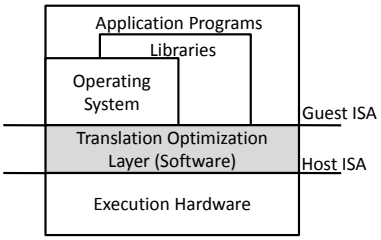

Fig. 1. HW/SW interface in co-designed processors

Software dynamic binary translation/optimization provides several benefits over the hardware implementation. For example, a hardware dynamic binary translator, typically, translates individual CISC instruction to microcode in isolation whereas, a software implementation translates the entire basic block or even bigger code region at once hence increasing the scope of optimizations. Furthermore, it allows to upgrade a processor in the field by introducing new optimizations or fixing bugs in the software layer which is not feasible with a hardware implementation. The software implementation of TOL also reduces validation and verification cost and time. Finally, the software implementation reduces hardware complexity.

A. Dynamic Binary Translation and Optimization

As said before, translating the guest ISA code to host ISA is the prime responsibility of TOL. The translation is done dynamically and generally, in multiple phases. Usually, in the first phase, an interpreter decodes and executes guest ISA instructions sequentially. In the rest of the phases, the guest code in translated to host ISA code and stored in a code cache, after applying several dynamic optimizations, for faster execution. The number of translation phases and optimizations in each phase are implementation dependent.

A typical two stage TOL starts by interpreting the guest ISA instruction stream sequentially. While interpreting, TOL also profiles the guest code to collect information about the most frequently executed code and biased branch directions. The execution frequency guides TOL to decide which guest code basic blocks to translate. When a basic block has been executed more than a predetermined number of times, TOL invokes the translator. The translator takes the guest ISA basic blocks as input, translates them to host ISA code and saves the translated code into the code cache for fast native execution. Instead of translating and optimizing each basic block in isolation, the translator uses biased branch direction information, collected during interpretation, to create bigger optimization regions, called superblocks. A superblock,

generally, consists of multiple basic blocks following the biased direction of branches. Therefore, superblocks increase the scope of optimizations to multiple basic blocks and allow more aggressive optimizations. Superblocks have a single entry point that is the first instruction of the first basic block included in the superblock. However, the number of exit points are implementation dependent.

III. Challenges and Design Choices

Following are the key challenges and design choices in HW/SW co-designed processors that need a simulation infrastructure, like DARCO, for evaluation :

Startup Delay: The time taken for initial translations before executing the translated/optimized native code, called startup delay, is an important factor that dictates the startup performance or response time of the system. Transmeta Crusoe's software layer starts with interpreting the guest code and deferring translations until the detection of hot code. Even though interpretation is cheaper than translation, it is still significantly slower than executing native code and hence, results in poor startup performance. This was also one of the major reasons for Crusoe's demise. Nvidia Denver employs a guest ISA decoder, in addition to the native decoder, to avoid interpreting/translating the cold code and thereby boosting the startup performance. However, a dual decoder system results in additional hardware complexity and power consumption. Reducing the startup delay without additional hardware complexity remains an open question.

When and where to translate/optimize: An interesting question is: Should the guest code be translated/optimized as soon as it becomes a candidate for optimizations or should it be deferred to possibly increase resource utilization. For example, in a multicore processor, it might be desirable to defer optimizations until a core becomes idle, so that optimizations can get under-way while the unoptimized code is executing on the original core in parallel. Similarly, how many, if any, cores should be dedicated to translation/optimization in a manycore system is also an interesting question.

Finding optimal instruction schedule: McFarlin et al. [14] showed that 88% of the performance gain of out-of-order (OoO) processors can be achieved by a single "best" static schedule. In a heterogeneous multicore system with a mix of in-order and OoO cores, how to profile the execution to find the "best" schedule on an OoO core and then migrate it to an in-order core is a compelling problem.

Speculative Execution: As co-designed processors avoid aggressive OoO execution, they rely heavily on aggressive code optimizations and speculative execution for performance. Speculative execution poses its own challenges. First of all, it requires to checkpoint the architectural state periodically to recover from a possible speculation failure. The checkpointing granularity presents an interesting trade-off, as a high granularity reduces checkpointing/commit overhead however, more work may need to be flushed on speculation failures. How to find susceptible speculation failure points to take a checkpoint just before them is an intriguing problem.

Detecting a speculation failure itself can be costly. For example, a common approach to detect speculative memory reordering failures is to store the addresses and sequence numbers of all the speculatively executed memory instructions in a hardware table and search the table to check if the current memory reference aliases with previously executed operations. Doing a parallel search in such a table consumes significant power or limits the table size. On the other hand, a serial search increases the latency.

Profiling: Co-designed processors employ binary instrumentation to profile execution behaviour of applications and optimize them accordingly. Profiling can be a powerful tool in detecting phase changes during program execution and re-tuning the code correspondingly. However, as instrumentation code executes every time along with the application code that it profiles, it can degrade the overall performance. A common approach to reduce this performance penalty is to avoid profiling the optimized code. However, not profiling the optimized code means losing the opportunity to track phase changes and re-tuning a major portion of the application as the optimized code accounts for more than 90% of execution time. How to strike a balance between profiling overhead and the performance improvement from re-tuning and what hardware support can accelerate profiling remain to be seen.

Wide in-order or narrow out-of-order cores: HW/SW co-designed processors, generally, employ in-order cores to reduce the energy requirements. However, an interesting trade-off would be to compare the performance/watt of wide in-order cores against narrow out-of-order cores taking into consideration the dynamic optimizations.

Interaction between TOL and application: TOL is invoked throughout the application execution for translation, optimization of the guest code or for guiding the application execution as such. As TOL and applications compete for the shared resources, it is important to understand their interaction and its impact on overall performance.

IV. Challenges in building a HW/SW co-design infrastructure

This section describes the key challenges and essential features that an infrastructure needs to provide to simulate a HW/SW co-designed processor. The bulk of complexity of such an infrastructure comes from the modelling of TOL. TOL not only needs to provide dynamic profiling, translation and optimization capabilities but also handles the corner cases that have been moved up from the hardware to optimize it for the common case, e.g. handling of complex string instructions might be moved to the software layer from the hardware. Following are the key features that a HW/SW co-design simulation infrastructure needs to provide:

Correctness: Dynamic binary translation and optimization lies at the heart of HW/SW co-designed processors. The software layer translates the guest binary to the host ISA and applies a number of aggressive, often speculative, optimizations. As the correct execution of the guest binary is the prime responsibility of a processor, the simulation infrastructure must provide ways

to validate the correctness of translation/optimization passes.

Minimum software layer overhead: The overall execution time in a HW/SW co-designed processor can be divided into two components: application execution time and TOL execution time. TOL execution time corresponds to the execution of TOL when it is doing translation and optimizations or otherwise guiding the application execution. Since during TOL execution time the application is not making any forward progress, it can be considered as overhead. It is of utmost importance to keep this overhead minimum because a high TOL overhead would negate the performance benefits of dynamic optimizations. From the simulation infrastructure point of view, a negligible TOL overhead is desirable as it will lead the application code to dominate the execution time and facilitate evaluating different optimizations as their contribution would be more pronounced.

Minimum emulation cost: Emulation cost is the average number of host ISA instructions required to emulate one guest ISA instruction. The emulation cost should be kept as low as possible. From the simulation infrastructure point of view, a high emulation cost will make it difficult to evaluate the real power and performance benefits of co-designed execution compared to the native execution as it would be executing a lot more instructions than the native execution.

Support for multiple guest ISAs: One of the distinguishing features of HW/SW co-designed processors is the ability to execute multiple guest ISAs on the same hardware. For example, Nividia Denver was initially rumoured to be an x86 CPU and later a dual mode design [15]. This feature is going to be especially important in future as one would like to be able to run any application (compiled for any ISA) on any computing device. Therefore, the simulation infrastructure needs to provide a flexible frontend interface to support multiple guest ISAs.

Plug-and-Play support: The simulation infrastructure needs to provide the ability to implement and evaluate new features such as new dynamic optimizations, new hardware units or hardware support for some optimization, host ISA extension, etc. Moreover, it is desirable that the integration of these new features to the existing infrastructure is simple and straightforward. Such a plug-and-play approach would allow researchers to utilize their time in coming up with innovative ideas rather than squandering it in figuring out how to integrate their techniques into the infrastructure.

Powerful debug toolchain: Given the complexity of simulating a HW/SW co-designed processor, the simulation infrastructure should provide a powerful debug toolchain especially compared to the conventional architectural simulators. In such an infrastructure an erroneous behaviour could be caused by a bug in distinct modules of TOL such as translator, optimizer, instruction scheduler, register allocator, code generator, etc. in addition to conventional functional and timing simulators. Therefore, a powerful debug toolchain becomes essential to quickly locate and fix any bugs.

V. DARCO Architecture

DARCO encompasses the guest and host ISA functional emulators, the software layer called Translation Optimization Layer (TOL), timing and power simulators and the accompanying debugging and monitoring tools. DARCO has a clean interface for including new optimizations in TOL and allows easy implementation of new hardware features. As shown in Figure 2 DARCO is composed of four main components:

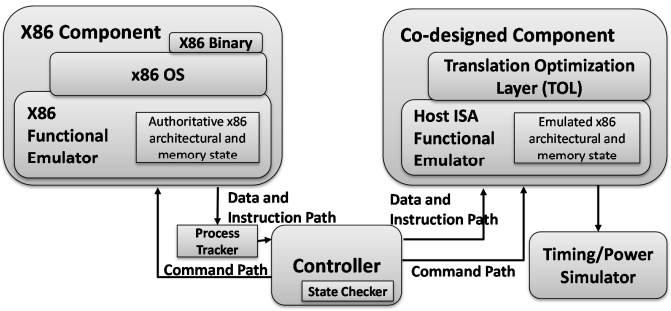

Fig. 2. DARCO main components

The ***co-designed component*** models the functionality of a HW/SW co-designed processor. It is constituted of TOL and a functional emulator for the host ISA. TOL translates and optimizes the guest x86 binary before the host functional emulator executes the optimized code. TOL itself is also compiled to the host ISA. This component keeps the emulated x86 architectural and memory states which are updated as the application execution proceeds. In its current state, the co-designed component models only the user level code.

The ***x86 component*** provides a full-system functional emulator for the guest x86 ISA. It runs an unmodified operating system and is the only component that interacts with the operating system. The authoritative architectural and memory state is also kept by this component as it emulates x86 code and not the translated/optimized code. The x86 component plays an instrumental role in ensuring the correctness of translation and optimizations in the co-designed component by validating the emulated architectural and memory states kept by the co-designed component against its own authoritative states. DARCO relies on the x86 component to execute the system code as the co-designed component models only the user code.

The ***timing simulator*** models a parameterized in-order core. The simple in-order processor is chosen in congruence with the simple hardware design philosophy of HW/SW co-designed processors. It receives the dynamic instruction stream from the co-designed component and provides detailed execution statistics. Furthermore, McPAT [8] has been integrated with the rest of the infrastructure for power and energy modelling. The use of the timing and power simulators is optional and does not affect the functionality of the rest of the infrastructure.

The ***Controller*** is the main user interface of DARCO. It provides full control over the execution of the application as well as debugging utilities. The main task of the controller is to provide synchronization among different components and the resolution of the various requests from the co-designed component as will be explained in next section. The controller is also responsible for comparing authoritative and emulated x86 states to ensure the correctness of translation/optimizations.

A. Execution Flow

The execution flow of an application passes through three distinct phases: 1) Initialization, 2) Execution, and 3) Synchronization. During the Initialization phase, the controller first starts the co-designed component, which in turn, initiates the execution of TOL. The co-designed component then remains idle until the controller sends the initial x86 architectural state of the application to be executed to it. As for the x86 component, when launched by the controller, it initiates the execution of the application defined by the user. When it reaches the system call EXECVE (which always takes place at the beginning of an application) the execution pauses. A process tracker is initialized with the application's Control Register 3 (CR3) value, which can be used to distinguish the specific process from the rest of the applications running on top of the operating system. After process tracker initialization, the x86 component sends the initial x86 state of the application to the controller. The Initialization phase is completed when the controller forwards this state to the co-designed component.

In Execution phase, TOL begins by executing code from the initial Program Counter it received during the Initialization phase. All changes made to the x86 architectural and memory state from the emulation of the x86 instructions are stored in the "Emulated x86 architectural and memory state". While the x86 application is making forward progress in the co-designed component, the x86 component remains idle.

The Synchronization phase is initiated by the co-designed component when any of the following three events occurs during the Execution phase: 1) data request, 2) system call, or 3) end of application. The data request event occurs when the co-designed component accesses an x86 memory page for the first time. The co-designed component sends a request to the controller for the particular memory page along with the total number of dynamic x86 basic blocks executed till that point. Then, it remains idle until the request is satisfied. The controller forwards the request to the x86 component, which in turn continues the execution of the application until it reaches the same execution point as the co-designed component (remember that the x86 component remained idle after the initial launch of the application). When the correct execution point is reached, the requested memory page is sent to the controller and forwarded to the co-designed component. The same process is followed for the other two events: system calls and end of application. System calls need synchronization because the co-designed component models only user-level code. Therefore, the system calls are executed only in the x86 component. Any changes made to the x86 architectural and memory state during the execution of system calls are passed to the co-designed component after the completion of the system call. As for the end-of-application, the synchronization is necessary to ensure that the execution of the application on the co-designed component was correct.

B. Translation Optimization Layer

Translation Optimization Layer (TOL) of DARCO is responsible for translating the unmodified guest x86 code to a PowerPC-like RISC host ISA. TOL does this translation in three different modes: 1) interpretation mode (IM), 2) basic block translation mode (BBM), and 3) superblock translation and optimization mode (SBM). TOL starts by interpreting the guest x86 code and promotes it to the higher optimization levels as hot code regions are detected. The high level view of TOL execution flow is shown in Figure 3.

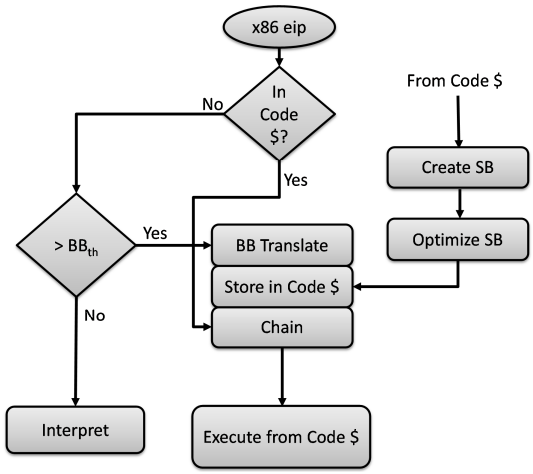

Fig. 3. TOL execution flow. The left path is followed in IM, the middle in BBM and the right in SBM

1) Interpretation: TOL begins the execution of the application in IM. While in IM mode, x86 instructions are interpreted one by one and the emulated x86 state is updated accordingly. The IM guarantees forward progress of the application and is also used as a safety-net in case instructions cannot be included in basic block translations and superblocks. Moreover, interpretation is necessary to make forward progress in case of speculation failures in superblock due to aggressive optimizations.

2) Basic Block Translation: During IM, profiling information is collected for execution frequency of the basic blocks using software repetition counters. When the repetition counter of a basic block reaches a predetermined threshold, TOL switches to BBM in order to translate the corresponding x86 basic block. In this mode, the guest x86 instructions are translated to intermediate representation (IR) code. The IR code then undergoes some basic optimizations like dead code elimination and constant propagation, which smoothens the rough edges caused by the translation of individual instructions. Finally, a code generator maps the optimized IR to host instructions and stores them in the code cache.

3) Superblocks and Optimizations: During Basic-Block translation Mode (BBM), profiling information is gathered for all the basic blocks using software counters. This information consists of execution and edge counters. The execution counters provide the execution frequency of the basic blocks while the edge counters monitor the biased direction of branches. Once the execution of a basic block exceeds another predetermined threshold, TOL creates a bigger optimization region, called superblock, using the branch profiling information collected during BBM.

In Superblock translation and optimization mode (SBM), TOL generates a new superblock starting from the triggering basic block. A superblock generally includes multiple basic blocks following the biased direction of branches, in other words speculating the control flow. A superblock ends at one of the following conditions: 1) The last basic block included in the superblock ends with an indirect branch, call, or return instruction; 2) The last basic block included in the superblock ends with an unbiased branch or the probability of reaching the last basic block from the beginning of the superblock falls below a predetermined threshold; 3) The number of instructions in the superblock exceeds a predetermined threshold; 4) The number of basic blocks included in the superblock exceeds a predetermined threshold.

Moreover, the branches inside the superblocks are converted to "asserts" so that a superblock can be treated as a single-entry, single-exit sequence of instructions. This gives the freedom to reorder and optimize instructions across multiple basic blocks. "asserts" are similar to branches in the sense that both check a condition. Branches determine the next instruction to be executed based on the condition; however, asserts have no such effect. If the condition is true, assert does nothing. However, if the condition evaluates to false, the assert "fails" and the execution is restarted from a previously saved checkpoint in IM. Furthermore, if the number of assert failures in a superblock exceeds a predetermined limit, the superblock is recreated without converting branches to "asserts". As a result, this time the superblock has to be treated as a single-entry multiple-exit sequence of instructions. Having multiple exits in a superblock also reduces available optimization opportunities because the instructions across different exit paths cannot be reordered as freely as before.

Furthermore, loop unrolling is also employed during superblock creation. Currently, we unroll loops consisting of only a single basic block, as they are the ones which provide maximum benefit [16]. Moreover, the unrolled version of the loop is followed by the original loop (without unrolling). During execution, a runtime check is performed to determine whether to execute the unrolled version or the original loop. The unrolled version is executed only if the number of iterations remaining are more than the loop unroll factor.

TOL optimizer provides a set of basic code optimizations to start with. Moreover, the modular optimizer design makes it convenient to enable/disable and plug-in new optimizations in TOL. The optimizer starts with transforming the intermediate representation (IR) code of a superblock into a Static Single Assignment (SSA) format. This transformation removes anti and output dependences and significantly reduces the complexity of subsequent optimizations. Second, a forward pass applies a set of conventional single pass optimizations: constant folding, constant propagation, copy propagation, and common subexpression elimination. Third, a backward pass applies dead code elimination.

After these basic optimizations, the Data Dependence Graph (DDG) is prepared. To create DDG, the input and output registers of the instructions are inspected and the corresponding dependences are added. Memory disambiguation analysis is also performed during DDG creation. If the analysis cannot prove that a pair of memory operations will never/always alias, it is marked as "may alias". In case of reordering, the original memory instructions are converted to speculative memory operations. Furthermore, Redundant Load Elimination and Store Forwarding are also applied during DDG phase so that redundant memory operations are removed. The DDG is then fed to the instruction scheduler that uses a conventional list scheduling algorithm. Afterwards, the determined schedule is used by the register allocator that implements Linear Scan Register Allocation algorithm. Finally, the optimized IR code is translated to the host code and is stored in the code cache. The previous entry in the code cache that corresponds to the first basic block of the current superblock is invalidated and freed for later translations.

C. Timing Simulator

DARCO models an in-order superscalar processor with front- and back-ends running independently and separated by an instruction queue. The front-end (equipped with a BTB and gshare branch predictor) fetches, decodes and stores the host instructions in the instruction queue. The backend issues and executes the instructions from this queue and employs Scoreboarding to track dependencies and resource availabilities. It incorporates simple, complex and vector units for execution. DARCO models two level TLB and cache hierarchies with a stride data prefetcher. Timing simulator parameters include: issue width, instruction queue size, numbers of execution units and latencies, number of physical registers (scalar/vector), branch predictor and BTB sizes, cache and TLB sizes/latencies, numbers of memory read/write ports and vector length for SIMD units.

D. Meeting the challenges

Having already described DARCO architecture, next we explain how it meets the challenges described in Section IV:

Correctness: The x86 component holds the key to ensure the correctness of the co-designed execution. It executes the unmodified x86 binary and keeps authoritative architectural and memory states. DARCO compares the architectural and memory states of x86 and co-designed components periodically to ensure correctness. The comparison kicks off by itself at system calls and at the end of application. Furthermore, the user can also decide how often to validate co-designed states against the authoritative x86 states.

Minimum TOL overhead: In addition to carefully engineering the TOL code itself, DARCO employs several other techniques to minimize TOL overhead. For example, 3-stage translation/optimization (IM, BBM and SBM) ensures that only the most frequently executed code is optimized aggressively, therefore reducing the optimization overhead. Furthermore, DARCO ensures that TOL is invoked only when absolutely necessary while executing optimized code from the code cache. To reduce the number of TOL invocations, for example, the optimized superblocks (and basic blocks) are chained together whenever possible. Even the indirect branches are chained using IBTC [17].

Fig. 4. Dynamic x86 instruction distribution in IM, BBM and SBM

Minimum emulation cost: The biggest factor in reducing the emulation cost is runtime optimizations. DARCO generates aggressively, if needed speculatively, optimized host code that not only improves performance but also lowers the emulation cost. Furthermore, DARCO utilizes several other techniques to ensure minimum emulation cost. For example, DARCO maps guest architectural registers directly on the host registers. Therefore, it saves upon the instructions that would otherwise be needed to load a register on reads and store it on writes if the guest architectural registers were kept in memory. Additionally, x86 instructions have a side effect of writing the flag register. DARCO writes to the flag registers only if the written value is really going to be consumed by a subsequent conditional instruction.

Support for multiple ISA: Even though DARCO currently supports only x86 as guest ISA, incorporating additional frontends is straightforward. As explained in Section V-B, TOL first translates the guest ISA instructions to an intermediate representation (IR) and all the optimization are applied to IR. To support additional guest ISAs, DARCO just needs another software decoder to translate the new guest ISA instructions to IR. All the following steps from SSA to code generation can be shared among the different guest ISAs.

Plug-and-play: DARCO is designed in a modular way making it straightforward to enable/disable any particular feature without affecting the functionality of the rest of the infrastructure. A clean interface makes it easy to develop new features, optimizations or hardware features, in isolation and plug them into the whole system when ready.

Debug toolchain: DARCO provides a strong debug toolchain with the x86 component playing a major role in it. As DARCO periodically validates the co-designed architectural and memory states against the authoritative x86 state, it activates the debug mechanism if a mismatch is detected. DARCO, first of all, pinpoints the exact basic block where the problem was originated. Then it traces back to find out the particular step where the bug first appeared, e.g. while translation to IR, any of the several optimizations, during emulation in the host ISA emulator, etc.

VI. DARCO Evaluation

This section presents a high level evaluation of DARCO and characterizes TOL using a set of benchmarks from SPEC2006 [18] and Physicsbench [19]. First, we examine the execution speed of DARCO followed by a discussion on dynamic x86 instruction distribution in different execution modes. Then a study of the emulation cost of x86 instructions in SBM is presented. Finally, we study TOL overhead and its components followed by a brief case study.

A. DARCO speed:

We present DARCO speed in terms of number of instructions emulated/simulated per second for both the guest and host ISA. For the guest x86 ISA, DARCO speed represents the rate at which the x86 instructions pass through the entire execution flow of DARCO including all the components shown in Fig 2. On average, DARCO emulates 3.4 million x86 instructions per second (MIPS) whereas the simulation speed with the timing simulator enabled is 370 KIPS for the guest ISA. Similarly for the host ISA, the emulation speed is 20 MIPS and the simulation speed with the timing simulator is 2 MIPS. The results presented are collected on a cluster where only one core is devoted per execution task and all the components of DARCO share this core.

B. Optimized Code Distribution:

The dynamic x86 instruction distribution in the three execution modes of TOL: the Interpreter Mode (IM), the Basic Block Mode (BM), and the Super Block Mode (SBM) is depicted in Figure 4. The experimental data shows that 88%, 96%, and 75% of the dynamic instruction stream comes from the highest level of optimization, superblocks, in SPECINT2006, SPECFP2006, and Physicsbench respectively. For three benchmarks in Physicsbench, namely *continuous*, *periodic*, and *ragdoll*, significant number of instructions are executed in BBM. The dynamic instruction count and dynamic to static instruction ratio for these benchmarks are significantly lower than other benchmarks. Therefore, only a small portion of code is promoted to SBM.

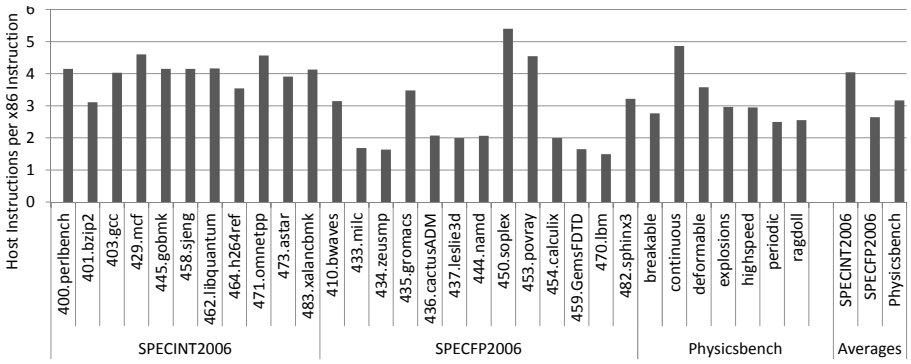

Fig. 5. Host instructions per x86 instruction in SBM

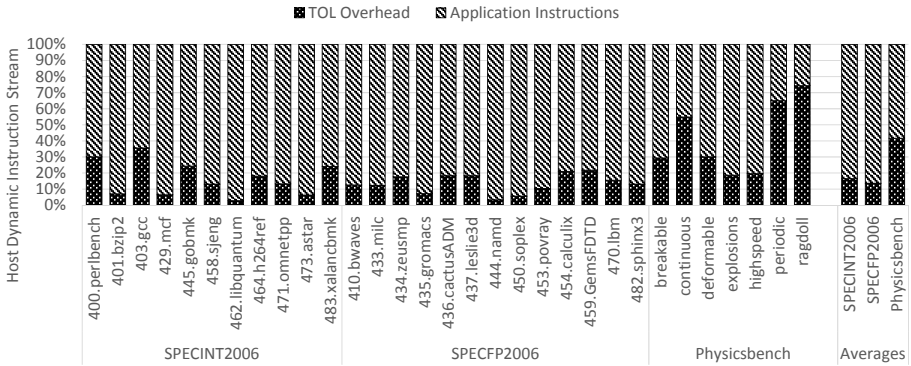

Fig. 6. Overall Host Dynamic Instruction Distribution

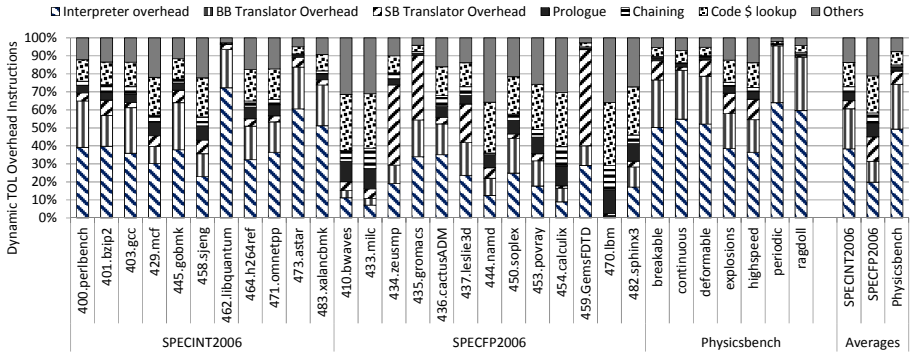

Fig. 7. Dynamic TOL Overhead Distribution

C. Emulation Cost:

Emulation cost is the number of host instructions generated per x86 instruction. Since the dynamic instruction stream is dominated by the execution of code in SBM, we present the emulation cost only in SBM. As Figure 5 shows, on average TOL generates 4, 2.6, and 3.1 host instructions per x86 instructions for SPECINT2006, SPECFP2006, and Physicsbench respectively. The emulation cost of SPECINT2006 is relatively high because of high emulation cost of branches. Since the basic blocks in SPECINT2006 are smaller, the emulation cost of branch instructions dominates the overall emulation cost. Physicsbench has a higher emulation cost because it uses significant amount of trigonometric functions like sin, cos, etc. These x86 instructions are not directly mapped to the host instructions, however, they are emulated in software. Therefore, the overall emulation cost increases.

D. Dynamic Instruction/Overhead Distribution:

The overall host dynamic instruction stream is composed of: 1) Application Instructions and 2) TOL Overhead Instructions. Application instructions are the dynamic instructions corresponding to the emulation of x86 application. On the other hand, TOL overhead are the instructions needed to translate the x86 code to the host code and doing housekeeping tasks.

Figure 6 shows the percentage of application instructions vs TOL overhead in the host dynamic instruction stream. For SPECINT2006 and SPECFP2006, 16% and 13% of the overall instruction stream corresponds to TOL overhead respectively, whereas for Physicsbench this number rises to 41%. The high dynamic to static instruction ratio causes TOL overhead to be amortized in SPEC2006. On the other hand, in Physicsbench the overhead in not amortized due to fewer executions of translated code.

Figure 7 shows the various components of TOL overhead. It is divided into seven major categories (bottom-up): 1) Interpretation Overhead: TOL overhead for interpreting the code before it is promoted to BBM. 2) BB Translator Overhead: TOL overhead for translating the basic block promoted to BBM. 3) SB Translator Overhead: TOL overhead for creating, translating, and optimizing the superblocks. 4) Prologue: Every time control is transferred between TOL and the translated code, a specific piece of code is executed to do some housekeeping stuff like stack management. Prologue overhead corresponds to executing this code. 5) Chaining: Different translated basic blocks and superblocks can be connected to each other in the code cache. To check whether chaining is possible and chaining the possible pairs constitutes "Chaining" overhead. 6) Code Cache Lookup: Every time that control is transferred to TOL, it checks whether the translation for next x86 basic block is already present in code cache or not. This lookup is termed as code cache lookup overhead. 7) Others: All other overheads like managing control flow in the main loop of TOL, collecting statistics, TOL initialization, etc. fall under this category.

It is interesting to note that, in Physicsbench, Interpretation Overhead and BB Translator Overhead dominate the overall overhead, whereas in SPECFP2006 these overheads are relatively smaller. The reason for this behaviour is once again the dynamic to static instruction ratio. SB Translator Overhead, where most aggressive/speculative optimizations are applied, is relatively smaller for both benchmark suites.

E. Case Study

We present a brief case study to demonstrate the efficacy of DARCO in HW/SW co-design research. The details of the study can be found in [20]. DARCO has already catalyzed the research in this domain as is evident from [20]–[28].

Warm-up Simulation Methodology: Sampling based simulations are used universally to reduce the simulation time with minimal loss of accuracy. Sampling relies on selecting a small, yet representative set of samples from the application for detailed timing simulations. Statistics are collected for each sample after warming-up the microarchitectural components for few million instructions. In the case of HW/SW co-designed processors, however, the software layer state also needs to be warmed-up in addition to the microarchitectural state. Our experiments, using DARCO, show that warming-up the TOL state only for a few million instructions leads to significant loss of accuracy. The penalty of inaccuracy in TOL state is much severe than that in the microarchitectural state. For example, a last-level cache miss due to a warm up inaccuracy in microarchitectural state results in a penalty of hundreds of cycles, whereas a code region translation/optimization due to inaccurate TOL profiler state costs thousands to tens of thousands of cycles. Therefore, TOL state needs to be highly accurate after the warm up period to avoid inaccuracies in the statistics. To obtain such high accuracy, we found that, the warm-up period needs to be 3-4 orders of magnitude longer than the warm-up period for conventional processors. Such a long warm-up period results in correspondingly longer simulation times.

To tackle this problem, we have developed a new warm-up methodology that downscales the promotion thresholds during the TOL warm-up phase to allow the code to be promoted to the higher optimization regions quickly and restores the original thresholds while collecting statistics. It provides an interesting trade-off between the scaling factor and the warm-up period length. For example, a small scaling factor and long warm up period provide high accuracy but long overall simulation time. On the other extreme, a high scaling factor and small warm-up period reduce the simulation time at the cost of low accuracy. We use a heuristic to predict the scaling factor and warm-up length for each sample such that the warm-up execution represents the authoritative (complete, without warm-up) execution faithfully. Our off-line heuristic correlates the execution distribution (basic block execution frequencies) of different configurations, with different scaling factors and warm-up lengths, to the execution distribution of authoritative execution and picks up the best match. Our technique reduces simulation cost by 65x on average with an error of 0.75%.

VII. Related Work

Dynamic binary instrumentation, translation and optimization, which is an elementary part of HW/SW co-designed processors, has also been of interest in itself in the last decade. Dynamic binary instrumentation frameworks like Pin [29] and Valgrind [30] have been used to develop powerful tools for program analysis, debugging, security, etc. Dynamic binary translators and optimizers have been implemented in both hardware as well as software. Intel microprocessors implement dynamic binary translation in hardware to translate CISC instructions into μops [2]. On the contrary, Apple used Rosetta [31], a software dynamic binary translator, to run PowerPC binaries on Intel processors. Similarly, Dynamo [32], DynamoRIO [33], IA-32 EL [34], and Strata [35] perform optimizations in a software layer running on top of operating system. All these optimizers apply only simple, low-cost optimizations in order to minimize the optimization overhead. On the contrary, hardware-only dynamic binary optimizers like RePlay [36] and PARROT [37] do not incur this optimization overhead as optimizations are performed in hardware and off-the-critical path. However, hardware-only dynamic binary optimizers pay the cost in terms of extra hardware complexity, area and power.

Computer architects rely heavily on simulation infrastructures as is evident from the diversity of simulation infrastructures available today. Architectural simulators have consistently grown in terms of capabilities and complexity. Earlier simulators like SimOS [38], RSIM [39] and SimpleScalar [40] targeted uniprocessor architectures. However, with the advent of parallel and multicore architectures a new class of simulation infrastructures like Simics [41], SIMFlex [42], GEMS [43], Graphite [44], Sniper [45], zsim [46], gem5 [47], ESESC [48], COTSon [49] etc. have taken over. Some of these simulators are able to simulate thousands of processor cores. Furthermore,

functional simulators and software virtualizers like QEMU [7], VMware [50], VirtualBox [51], etc. provide fast and flexible architectural emulation.

As we have seen in this section, there exist simulation infrastructures for dynamic binary translation, dynamic binary optimizations and microarchitectural simulations. However, there is no single infrastructure that provides all these features. Using an ad-hoc approach of combining these infrastructures to simulate a HW/SW co-designed processor is not attractive as all these infrastructures target different problems. For example, the dynamic binary optimizers discussed above optimize binaries assuming same guest and host ISAs which is not the case for HW/SW co-designed processors. More importantly, as these frameworks are developed in isolation without considering the synergy needed between the software layer (TOL) and the hardware, the ad-hoc approach would just try to make two disjoint components communicate with each other, whereas in HW/SW co-designed processors TOL and hardware are tightly coupled and specialized to work with one another. Therefore, we decided to develop the whole new simulation infrastructure instead of just putting few unrelated frameworks together.

VIII. Conclusion

This paper discussed the potentials of HW/SW co-designed processors and the challenges they need to address in order to compete with conventional hardware-only processors. We motivated the need of a simulation infrastructure and showed how it can be instrumental in exploring this domain and evaluating the design choices. Furthermore, we presented the challenges in building such a research infrastructure itself. In addition, we presented DARCO, a complete simulation infrastructure that provides host and guest ISA functional emulators, a translation optimization layer, a timing and power simulator for host ISA, a powerful debugging tool-chain, and monitoring tools. DARCO executes 90% of the guest dynamic code in the highest optimization level with minimal software layer overhead and incurs minimal emulation cost.

Acknowledgments

This work was supported by the Spanish State Research Agency under grants TIN2013-44375-R and TIN2016-75344-R (AEI/FEDER, EU).

References

[1] K. Olukotun *et al.*, "The case for a single-chip multiprocessor," in ASPLOS, 1996.
[2] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., 2005.
[3] D. Boggs, G. Brown, N. Tuck, and K. Venkatraman, "Denver: Nvidia's first 64-bit arm processor," *Micro, IEEE*, vol. 35, pp. 46–55, Mar 2015.
[4] A. Klaiber, "The technology behind the crusoe processors," in *White paper*, January 2000.
[5] S. S. *et al.*, "Boa: Targeting multi-gigahertz with binary translation," in *In Proc. of the 1999 Workshop on Binary Translation*, 1999.
[6] K. Ebcioğlu *et al.*, "Daisy: Dynamic compilation for 100architectural compatibility," in ISCA, 1997.
[7] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.
[8] S. Li *et al.*, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in MICRO, 2009.
[9] J. Dehnert *et al.*, "The transmeta code morphing™ software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in CGO, 2003.
[10] *Intel HW/SW co-designed processor project.* http://www.eetimes.com/document.asp?doc_id=1266396.
[11] M. Lupon *et al.*, "Speculative hardware/software co-designed floating-point multiply-add fusion," in ASPLOS, 2014.
[12] W. Cheng *et al.*, "Acceldroid: Co-designed acceleration of android bytecode," in CGO, 2013.
[13] N. Neelakantam *et al.*, "A real system evaluation of hardware atomicity for software speculation," in ASPLOS, 2010.
[14] D. McFarlin *et al.*, "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?," in ASPLOS, 2013.
[15] *Linley Group Microprocessor Report. Nvidias First CPU Is a Winner.* http://www.linleygroup.com/mpr/article.php?id=11262.
[16] S. S. Muchnick, *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
[17] K. Scott *et al.*, "Overhead reduction techniques for software dynamic translation," in *Parallel and Distributed Processing Symposium*, 2004.
[18] *Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks.* http://www.spec.org/cpu2006/.
[19] T. Yeh *et al.*, "Parallax: An architecture for real-time physics," in ISCA, 2007.
[20] A. Brankovic *et al.*, "Warm-up simulation methodology for HW/SW co-designed processors," in *CGO*, p. 284, 2014.
[21] R. Kumar *et al.*, "Assisting static compiler vectorization with a speculative dynamic vectorizer in an hw/sw codesigned environment," *ACM Trans. Comput. Syst. (TOCS)*, vol. 33, Jan. 2016.
[22] R. Kumar *et al.*, "Vectorizing for wider vector units in a HW/SW co-designed environment," in *HPCC*, pp. 518–525, 2013.
[23] J. Cano *et al.*, "Quantitative characterization of the software layer of a HW/SW co-designed processor," in *IISWC*, pp. 138–147, 2016.
[24] R. Kumar *et al.*, "Efficient power gating of SIMD accelerators through dynamic selective devectorization in an hw/sw codesigned environment," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 11, July 2014.
[25] A. Brankovic *et al.*, "Accurate off-line phase classification for HW/SW co-designed processors," in *Computing Frontiers*, pp. 5:1–5:10, 2014.
[26] R. Kumar *et al.*, "Speculative dynamic vectorization to assist static vectorization in a HW/SW co-designed environment," in *HiPC*, 2013.
[27] R. Kumar *et al.*, "Dynamic selective devectorization for efficient power gating of SIMD units in a HW/SW co-designed environment," in *SBAC-PAD*, pp. 81–88, 2013.
[28] R. Kumar *et al.*, "Speculative dynamic vectorization for HW/SW co-designed processors," in *PACT*, pp. 459–460, 2012.
[29] C. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in PLDI, 2005.
[30] N. Nethercote *et al.*, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in PLDI, 2007.
[31] *Apple Rosetta.* http://apple.wikia.com/wiki/Rosetta.
[32] V. Bala *et al.*, "Dynamo: A transparent dynamic optimization system," in PLDI, 2000.
[33] D. Bruening *et al.*, "An infrastructure for adaptive dynamic optimization," in CGO, 2003.
[34] L. Baraz *et al.*, "Ia-32 execution layer: A two-phase dynamic translator designed to support ia-32 applications on itanium-based systems," in MICRO, 2003.
[35] K. o. Scott, "Retargetable and reconfigurable software dynamic translation," in CGO, 2003.
[36] S. Patel *et al.*, "replay: A hardware framework for dynamic optimization," *IEEE Trans. Comput.*, vol. 50, June 2001.
[37] Y. Almog *et al.*, "Specialized dynamic optimizations for high-performance energy-efficient microarchitecture," in CGO, 2004.
[38] M. Rosenblum *et al.*, "Complete computer system simulation: The simos approach," *IEEE Parallel Distrib. Technol.*, vol. 3, Dec. 1995.
[39] C. Hughes *et al.*, "Rsim: simulating shared-memory multiprocessors with ilp processors," *Computer*, vol. 35, pp. 40–49, Feb 2002.
[40] T. Austin *et al.*, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, Feb. 2002.
[41] P. Magnusson *et al.*, "Simics: A full system simulation platform," *Computer*, vol. 35, Feb. 2002.
[42] T. Wenisch *et al.*, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, July 2006.
[43] M. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, Nov. 2005.
[44] J. Miller *et al.*, "Graphite: A distributed parallel simulator for multicores," in HPCA, 2010.
[45] T. Carlson *et al.*, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in SC, 2011.
[46] D. Sanchez *et al.*, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in ISCA, 2013.
[47] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, Aug. 2011.
[48] E. Ardestani *et al.*, "Esesc: A fast multicore simulator using time-based sampling," in HPCA, 2013.
[49] E. Argollo *et al.*, "Cotson: Infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, Jan. 2009.
[50] B. Walters, "Vmware virtual platform," *Linux J.*, vol. 1999, July 1999.
[51] J. Watson, "Virtualbox: Bits and bytes masquerading as machines," *Linux J.*, vol. 2008, Feb. 2008.